SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 389–419 (1996)

Research

A Toolset for Maintaining Hybrid C++ Programs

PANAGIOTIS K. LINOS
Computer Science Department, Tennessee Technological University, Box 5101, Cookeville, TN 38505, U.S.A.
VINCENT COURTOIS
Hautes Etudes Industrielles, 13 rue de Toul, 59046 Lille, France

SUMMARY

We present a toolset for maintaining C++ programs which are written using a hybrid object-orientated programming style (i.e., one that combines procedural and object-orientated techniques). The toolset maintains a database with control and data flow information found in source code and it is based on a compact hybrid data model for C++ programs. This information is visualized and manipulated both textually and graphically. A maintenance exercise performed on a C++ program using the toolset demonstrates that its code visualization features, abstraction mechanisms and graph management techniques constitute a promising platform towards the comprehension and maintenance of complex hybrid C++ code.

KEY WORDS: code visualization; data flow; control flow; inheritance hierarchy; file dependencies; colonnade; C++

1. INTRODUCTION

Software maintenance refers to any intentional modifications made to existing software (Chapin, 1988). It has been estimated that within the next decade we will be spending over a trillion dollars on maintaining existing software systems if we continue to use the same techniques, methods and tools used today (Edelstein, 1993). Moreover, many believe that this is happening mainly due to the difficulties related to the comprehension of large and complex computer programs. Various estimates indicate that approximately 50–70% of the total effort and time spent during software maintenance is devoted to program comprehension related activities (Sharon, 1996). Some researchers have proposed various clustering methods in order to support high-level program understanding (e.g., Choi and Scacchi 1990). However, very few of these ideas have been implemented in the context of a toolset. In particular, maintaining object-orientated code becomes a difficult problem due to some new features (e.g., inheritance, polymorphism) introduced by hybrid programming languages such as C++.

These features might offer flexibility; however, some complications during software maintenance can arise (Wilde and Huitt, 1992). For instance, the use of *classes* and *inheritance* often leads to a plethora of small program components (e.g., *objects*) with

CCC 1040-550X/96/060389-31

Received 13 March 1996
Revised 26 August 1996

many relationships (e.g., *message passing*) (Taenzer, Ganti and Podar, 1989). Today, there are several commercial and academic tools available which facilitate the development and maintenance of object-orientated programs. Examples of commercial tools include the *ObjectCenter* by Centerline Software Inc., the *ObjectWork* by Parcplace Systems and the *Imagix 4D* from the Imagix Corporation. In addition, several research prototype tools for maintaining object-orientated programs are available today. A software toolset described in Lejter, Meyer and Reiss (1992) entails a relational database with an interactive interface which supports queries about programs written in object-orientated programming languages. Another software tool provides browsing features through the source code of object-orientated programs using hypertext techniques (Sametinger, 1990). CIA++ is a stand-alone program analysis tool which provides extensive coverage of program dependencies (Chen and Grass, 1990). Finally, some visualization mechanisms for maintaining object-orientated software are described in De Pauw *et al.* (1993) which are based on a language-independent approach.

In general, software tools such as the ones mentioned above, facilitate maintenance activities on programs written using object-orientated programming languages by systematically extracting control and data flow information from source code. This information is stored in a database and then visualized in various graphical or textual representations. Some of the limitations of these tools include difficult to understand displays, limited abstraction mechanisms, and restricted ways for managing and browsing through large graphical representations. In particular, there is very little attention given to the problem related to visualizing code written using a mixture of procedural and object-orientated programming techniques. In this paper, we address the above issues by developing a prototype toolset for maintaining hybrid C++ programs. Finally, our research efforts are based on the general hypothesis that effective code visualization techniques can increase the productivity of the tool users and lead into better quality changes made during software maintenance (Linos *et al.*, 1994).

The next section of this paper presents an overview of the toolset. The section after that describes a maintenance exercise on a C++ program. Finally, we conclude with a summary and some suggestions for future work.

2. THE TOOLSET

2.1. A hybrid data model for C++ programs

Our toolset combines features of two previous prototype tools described in Linos and Courtois (1994) and Linos *et al.* (1993). The original functionality and GUI (graphical user-interface) of these tools are extended in order to facilitate the comprehension and maintenance of hybrid C++ programs. The GUI entails specially designed browser windows. For example, Figure 1 shows a screen of the toolset which includes its main window, with two graphical and one textual browser.

The toolset detects and populates the toolset database with a mixture of procedural and object-orientated program components from the C++ code. These components include *files* (both source and include files), *data-types* (both standard and user defined), *functions* (i.e., user defined functions), *constants*, *variables*, *parameters* (i.e., formal), *classes*, *objects* (i.e., instances of a class), and *member functions* (methods). In addition, various relation-

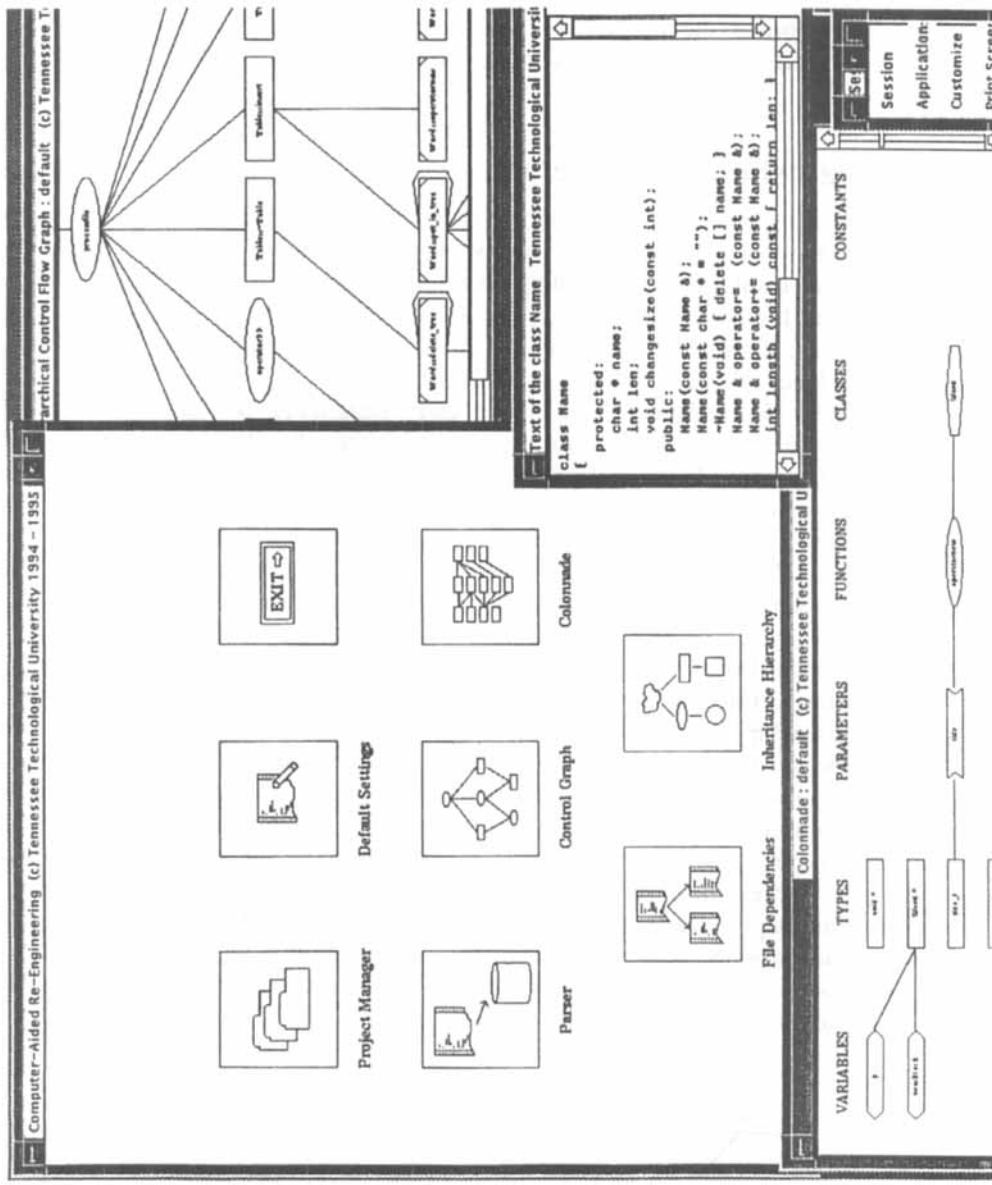

Figure 1. Example of the main window of the toolset, with two graphical browser windows and one textual browser window present

ships between these components are also included in the data model. These relationships entail the following; files *include* other files, classes *implement* (i.e., define) methods and *inherit* properties from other classes. Constants, variables and parameters *are defined as* of a type, objects *are instances of* classes. Moreover, functions may *return a value* of a type, *use* constants, variables and objects and they *send* messages to methods. Also, functions can *have* formal parameters, and they *call* other functions. Finally, methods *overwrite* other methods, *send* messages to other methods and *return* values of a class or a type.

2.2. Visualizing hybrid C++ programs

The control and data flow information included in the database can be displayed using a code visualization model designed specifically for understanding hybrid C++ code. It includes hierarchical displays for presenting class inheritance, control-flow information (e.g., call graph) and file dependencies (i.e., include files). In addition, an original display called *colonnade* (i.e., a sequence of columns displayed at regular intervals) is utilized in order to present object-orientated data-flow information (e.g., a class implements some methods) graphically. Such displays are created and manipulated by the use of several graphical editors implemented in the toolset. These include the *data-flow*, *control-flow*, *inheritance-hierarchy,* and *file-dependencies* graphical editors. Textual information (i.e., code) can also be manipulated in the toolset using a traditional text editor (e.g., emacs, vi). The control-flow graph editor displays user-defined functions as well as methods (i.e., member functions) with their relationships. Different graphical notations are used in order to distinguish between methods and functions. Specifically, function calls or message passing is denoted by connecting arrows between functions and methods. Figure 2 demonstrates how hybrid control-flow information is visualized within the same graphical display. A connecting arrow between two functions (or between a method and a function) depicts a call relationship as shown in Figure 2 (a). In Figure 2 (b), a connecting arrow between two methods (or a function and a method) represents a message being passed.

Moreover, the toolset addresses the difficulty of displaying information that can be determined only at run time (i.e., delayed binding). Specifically, information regarding *polymorphic* entities (e.g., virtual functions) is also displayed in the control-flow and the colonnade graphs. For example, in the case of a message passed to a *virtual* member function (where binding is delayed until run time) the toolset determines all possible functions where that message could be sent. Then, all the possible message paths for that virtual function can be displayed. A more detailed dicussion on this operation can be found in Linos and Courtois (1994).

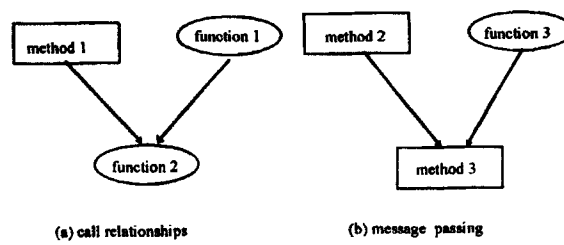

Figure 2. Visualizing hybrid control-flow found in C++ programs

The data-flow of hybrid object-orientated code is displayed separately. As mentioned earlier, this is accomplished by using an extended version of a graphical display called *colonnade* (Linos *et al.*, 1993). Experimental data have shown that the colonnade display appears to be a promising graphical model for visualizing data-flow information found in C programs (Linos *et al.*, 1994). It produces crossing-free (some replication of icons is necessary), easy-to-draw and aesthetically pleasing layouts. The toolset extends the colonnade in order to display information related to both procedural and object-orientated program components. The relationships between these components are represented by connecting lines between columns.

Figure 3 gives an example of a colonnade display of a C++ program produced by the toolset. In this figure we can see that the first column displays the variables (or objects) of the program and the second column depicts the data-types. The next column contains the parameters and the following one embodies the functions (i.e., user-defined or member functions). Finally, the fifth column displays the classes and the last one entails the constants of the program. Among others, Figure 3 depicts two constructor member functions for the classes *student* and *teacher* respectively. The *main* function is also displayed in the colonnade which uses an integer variable called *i* and six objects named *aperson, myperson, ateacher, myteacher, astudent* and *mystudent.* These objects are instances of their corresponding classes. Also, the *main* function references a variable called *list* which is an array of pointers to a *person* object. In addition, *main* utilizes a constant called *arraysize.*

The inheritance graph includes all the classes defined in a C++ program and their inheritance relationships (i.e., both *single* and *multiple* inheritance). Figure 4 displays the inheritance graph for a C++ program produced by the toolset. It shows that the *circle* and *square* classes both inherit from the *closed_figure* class. Moreover, the *cylinder* class inherits from *circle* whereas the *sphere* class inherits from both *circle* and *threeDobject* (i.e., *multiple* inheritance). Also, *cube* inherits from *square* and *threeDobject.* Moreover, *closed_figure* and *threeDobject* are *base classes* because they have no parent in the hierarchy. Finally, the file dependencies editor presents the *include* relationships between files using a hierarchical graphical display. Each file is represented by a node and included files are linked via connecting lines. Among others, we can see in Figure 5 that three source code files namely *main3a.c, main3b.c* and *table.c* all include a header file called *table.h.*

2.3. Managing complex representations

Large and complex program representations (both graphical and textual) become difficult to handle, comprehend and maintain. For graphs, this is happening mainly due to the large number of nodes and crossing lines cluttered on the screen. Some of our practical solutions to this problem entail techniques for breaking down large displays into smaller manageable pieces, abstraction mechanisms that remove unnecessary details and finally graph rearrangement operations which produce different *layout views* of the database.

2.4. Layout views

Large monolithic C++ programs can be decomposed into graphical and/or textual *layout views.* A *graphical* layout view is a fragment of the call-graph or the colonnade and a

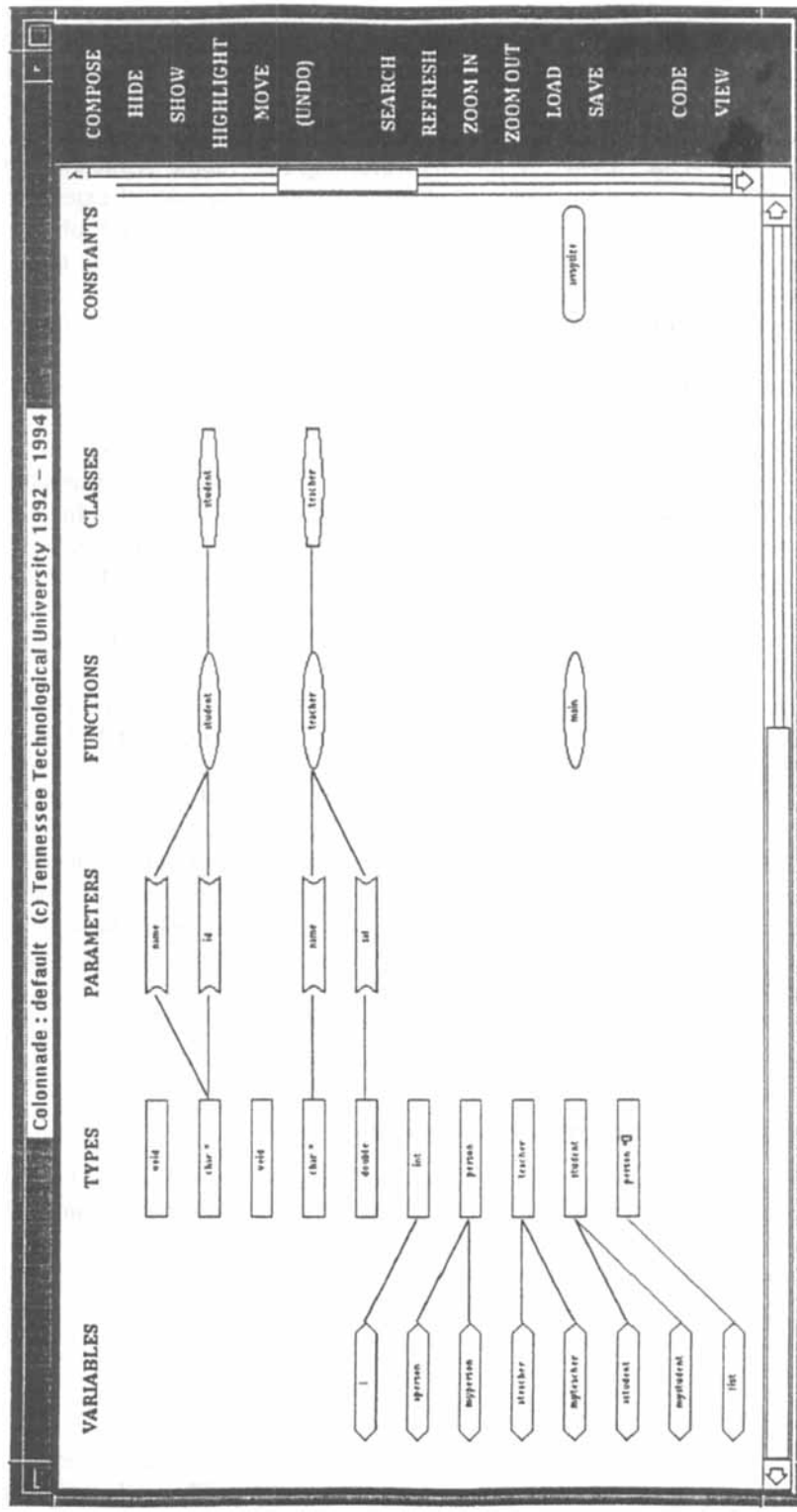

Figure 3. Example of a colonnade display by the toolset

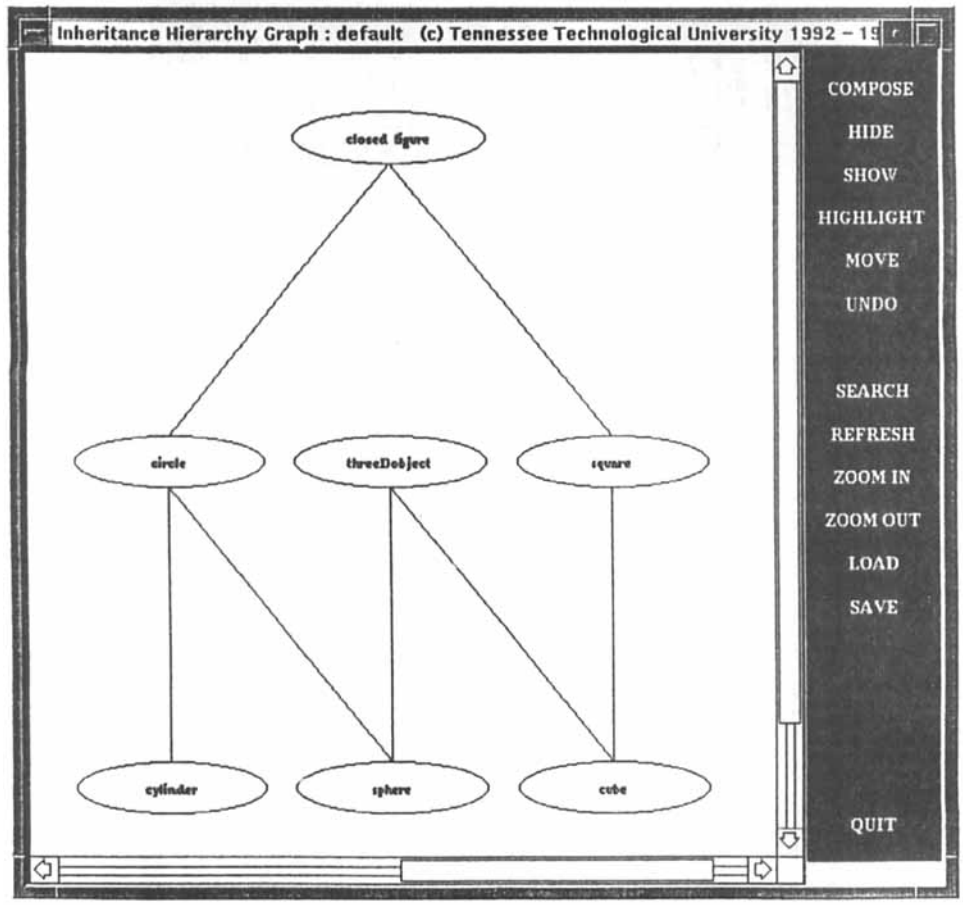

Figure 4. Example of an inheritance hierarchy graph in a C++ program

textual layout view refers to the body (i.e., code) of an entity. The rationale behind implementing *layout views* is to provide a means for partitioning large size graphs (or code) into manageable sub-graphs (or pieces of code).

The toolset supports two types of graphical layout views; the *data-flow* and *control-flow* views. The data-flow view is a portion of the colonnade in which all the entities are related to a single function. Similarly, a control-flow layout view is a two-level portion of the call-graph which includes the *calling* and *called-by* functions of a single *focus* function. Finally, a textual view displays only the code of a program entity (e.g., body of a function, definition of a data item, formal parameter, etc.). As an example, we can see three layout views for a function called *processfile* shown in Figure 6. Specifically, the data-flow layout view shows all variables and parameters used by this function along with their data types. The control-flow layout view displays all the messages being passed by *processfile* to various member functions as well as the fact that it is called by the *main* function. Finally, the textual view includes the body (i.e., code definition) for that function.

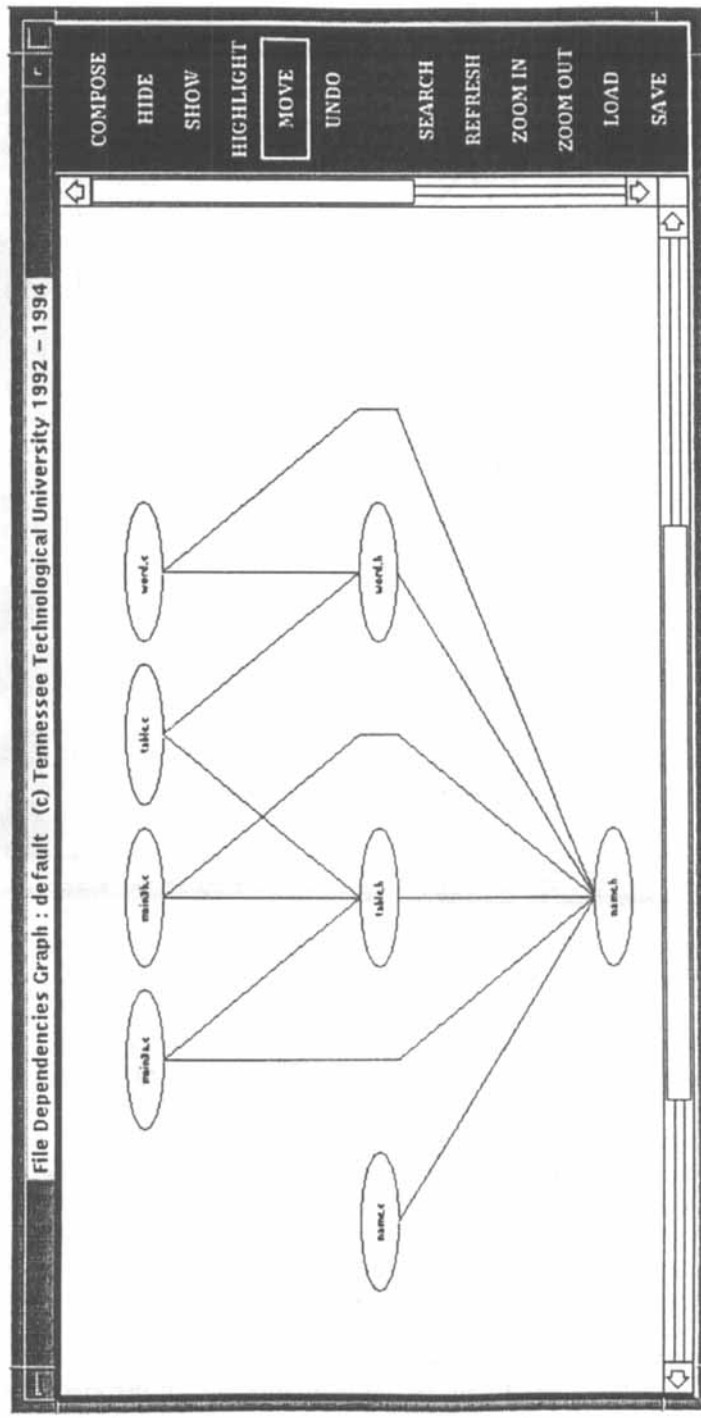

Figure 5. A graphical representation of file dependencies

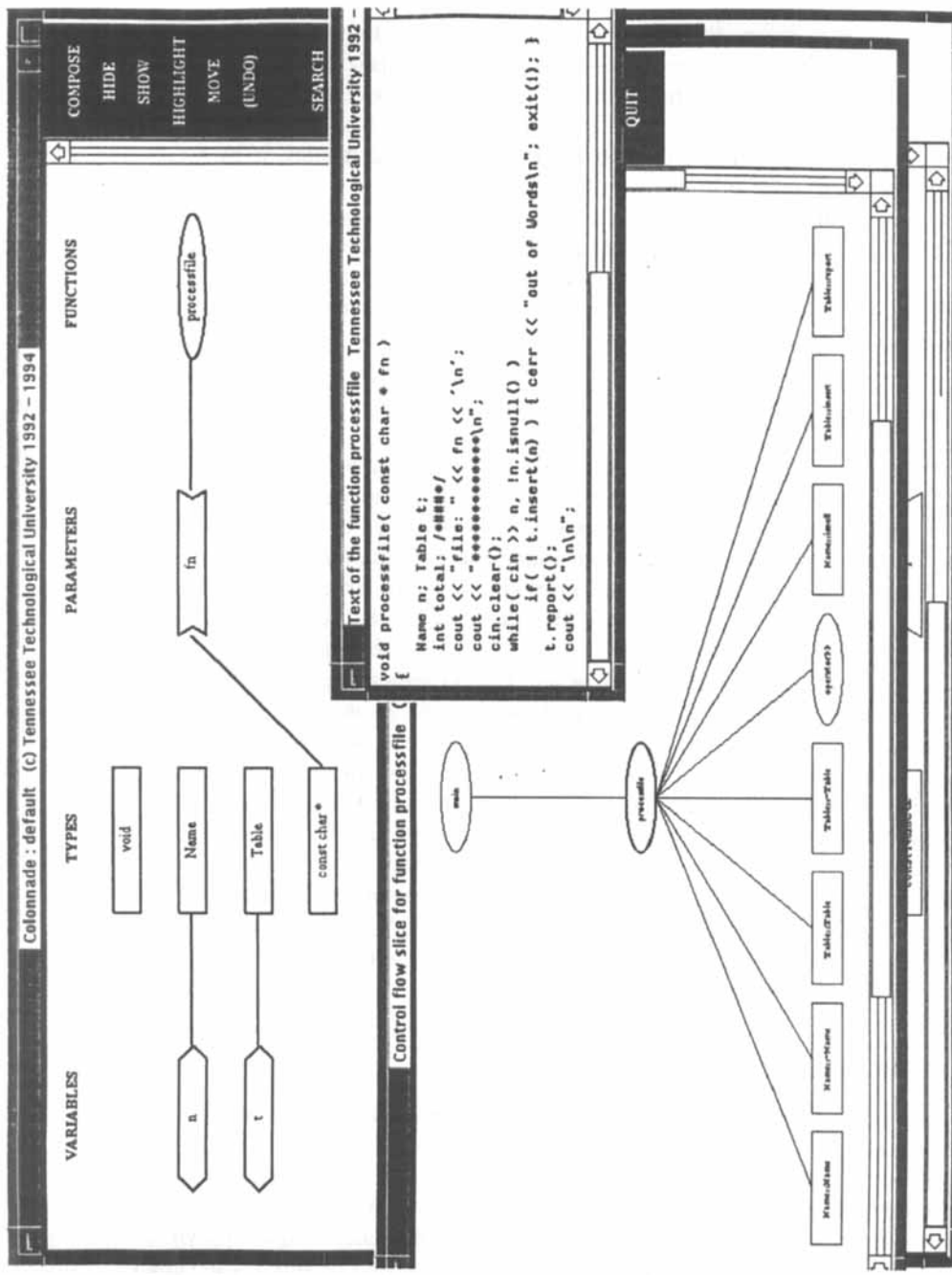

Figure 6. Three views for the function 'processfile'

2.5. Browsing

Each browser window is equipped with a group of operations for navigating among different kinds of code representations (i.e., text, graph). More specifically, the *code* operation enables the user to have direct access to the code from any graphical representation using a selected text editor (e.g., emacs, vi). The *search* feature allows the user to locate a desired entity in a graphical representation (and display it in the middle of the screen). The *view* operation allows the user to have access to data-flow or control-flow layout views from the call-graph or the colonnade graph respectively. This option facilitates *incremental browsing* in the database and it is accomplished by going directly from one layout view to another. Additional operations allow the user to *save* or *load* specific hierarchical or colonnade layouts. Each of the available operations can be cancelled by selecting the *undo* option.

2.6. Graph rearrangement

In order to enhance the management of complex graphical displays we implemented various operations for manipulating graphs and their components. Such operations provide the user with a means of creating customized and more legible graph layouts. For instance, graphs can be *scaled down* in order to make the information more visible, or *scaled up* to provide a more global view of the information shown. In addition, graphical nodes can be *hidden*, *highlighted* or *moved* to another location in the layout. A group of nodes can also be *composed* into a condensed node (a graphical abstraction technique).

In the case of colonnade layouts when two columns are not adjacent, the relations between their entities are not explicitly shown (a design decision that we made in order to avoid cluttered layouts). However, such relations can be displayed indirectly by using a *move-column* operation. Such an operation allows the user to change the positions of the columns in the colonnade. For example, if the relationships between functions and types need to be visualized, the move-column option can be used to create a different layout. Figure 7 shows the resulting colonnade layout after moving the FUNCTIONS column next to the TYPE column from the graph shown in Figure 3. We can now see relations that were hidden in the previous display such as the fact that *main* returns an integer value or that the classes *student* and *teacher* implement constructors which do not return any values (i.e., void).

2.7. Layout algorithms

In order to further improve the legibility of the graph drawings we have used variations of existing layout heuristics (Linos, Rajlich and Korel, 1991). Specifically, we have implemented two separate layout algorithms; one for improving hierarchical drawings (e.g., call graphs) and another for colonnade layouts. The hierarchies are produced using *polyline* drawings in order to avoid *bypassing* edges (i.e., edges drawn between non-adjacent levels), and the algorithms work even for ill-structured call graphs. Polyline connecting edges utilize dummy nodes which do not appear in the final layout. Figure 8 shows a hierarchical layout drawn with five dummy (tacit) nodes at locations (3,c), (2,d), (2,f), (1,f) and (1,g) in the matrix.

The hierarchical layout algorithm consists of five phases. Initially, all the nodes are

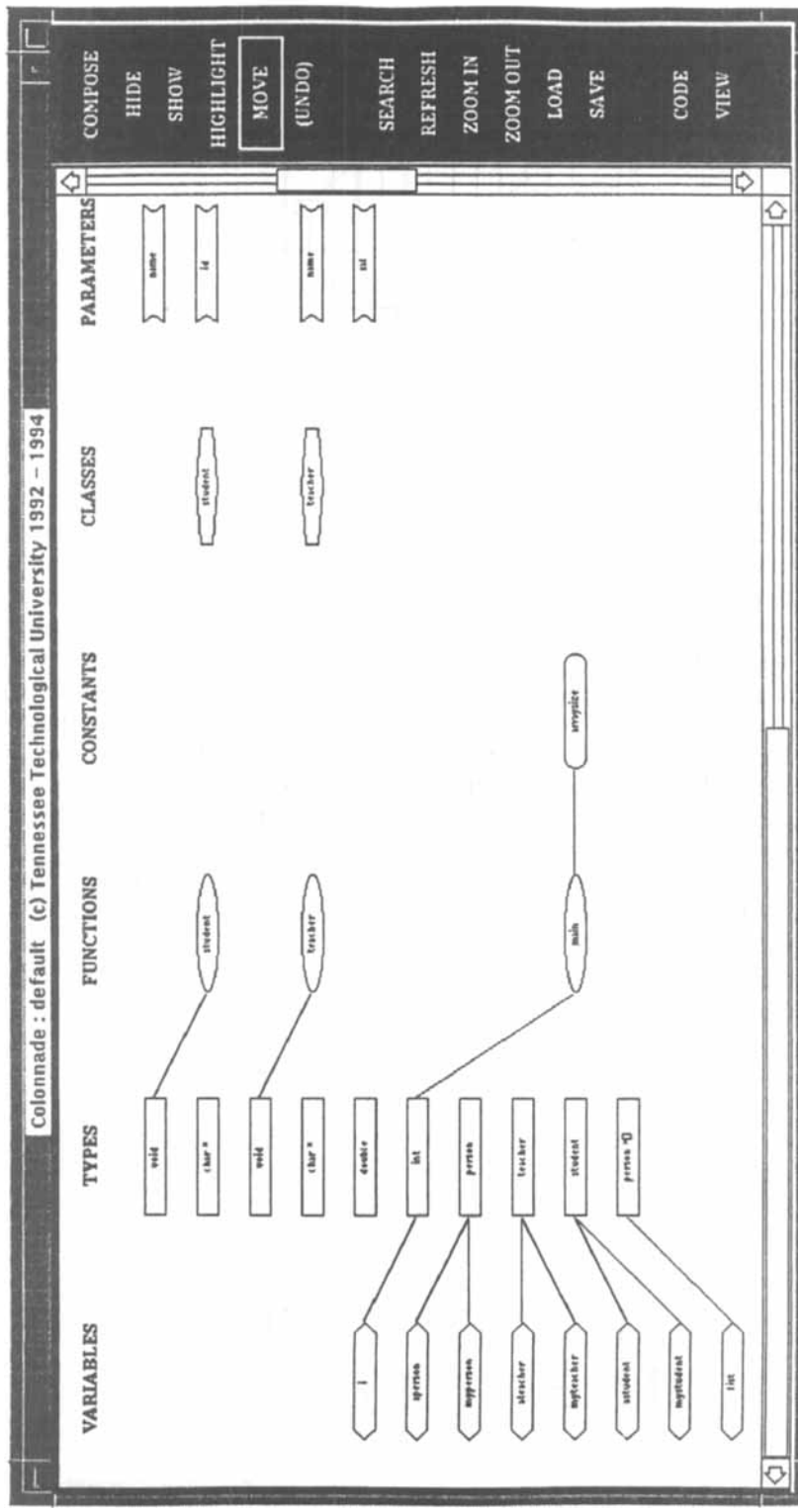

Figure 7. A different colonnade layout for the same program

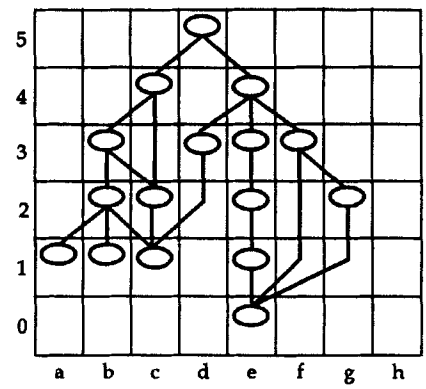

Figure 8. An example of a hierarchical call graph with polyline edges

assigned to the lowest level of the hierarchy (see Figure 9). The second phase determines the level for each node in the hierarchy. This is accomplished by looking at the nodes at the lowest level (i.e., leaf nodes). If one of these nodes is connected with another node in the same level, it is moved to the level immediately above. This process is repeated until no more nodes can be moved higher in the hierarchy (see Figures 10, 11 and 12). During the third phase, the algorithm looks at all leaf nodes. If a leaf node is connected with a function f which is positioned at level p, then f is repositioned at level $p-1$. For example, in Figure 13 node *f9* is placed at level 1 because it is connected with node *f3*. The fourth phase creates dummy nodes in order to handle bypassing edges. In Figure 13 *f6* in level 2 is connected with *f7* in level 0, a dummy node is inserted in level 1 so that the connecting line going from *f6* to *f7* does not cross over any other level (see Figure 14). Finally, the algorithm sorts the nodes on each level in order to reduce crossings of edges between levels. The algorithm calculates the new positions of the nodes on each level by using the mean of the positions of their adjacent vertices. Figure 15 illustrates the effect of that phase.

The algorithm for constructing the colonnade layout follows a layered approach. It constructs one *layer* (i.e., layout view) at a time (the colonnade can be thought of as the

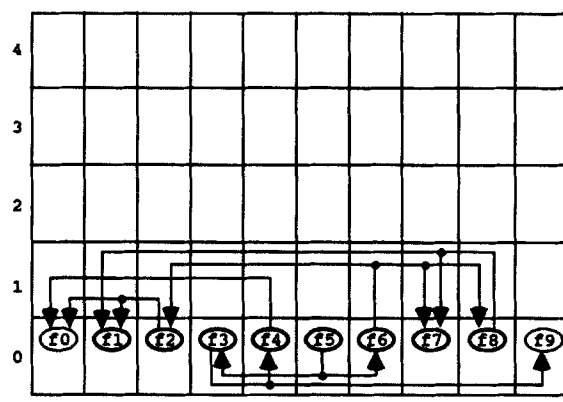

Figure 9. The hierarchical call graph after the first phase of the layout algorithm

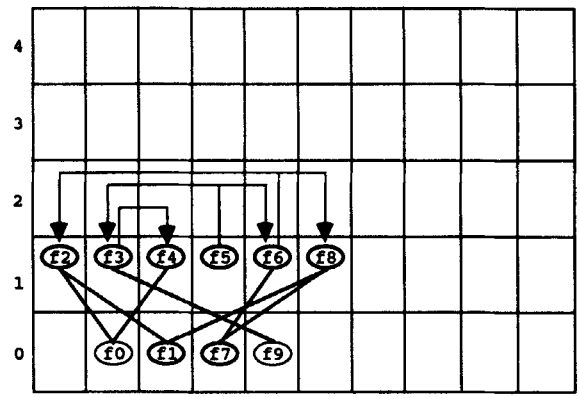

Figure 10. A second level is added in the hierarchy after the first iteration of the second phase

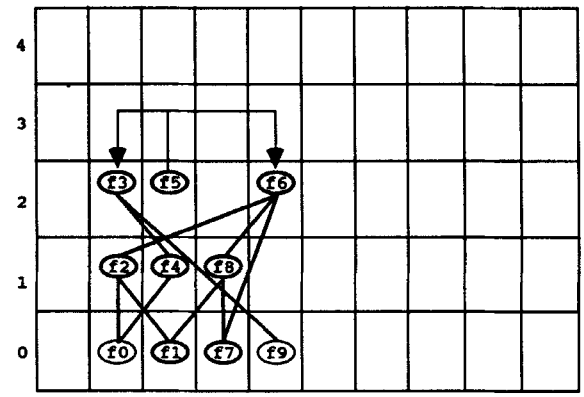

Figure 11. A third level is added in the hierarchy after the second iteration of the second phase

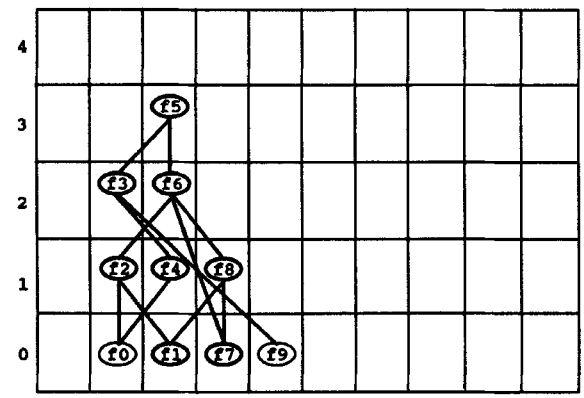

Figure 12. A fourth level is added in the hierarchy after the third iteration of the second phase

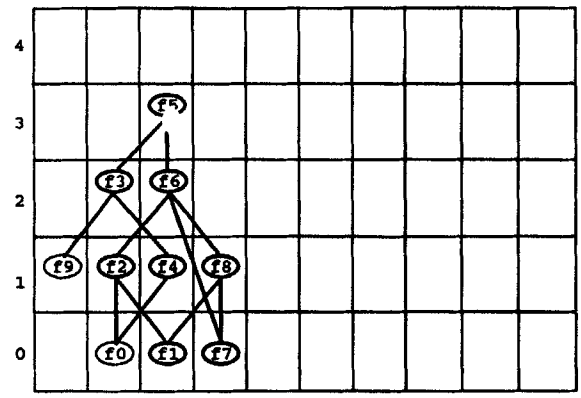

Figure 13. After the third phase, function f9 *is moved from level 0 to level 1*

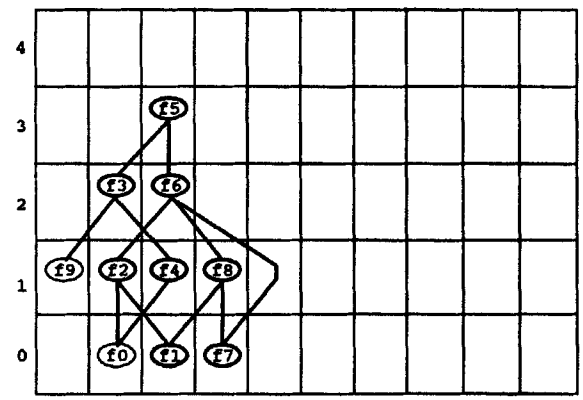

Figure 14. After the fourth phase, a dummy node is inserted between functions f6 *and* f7

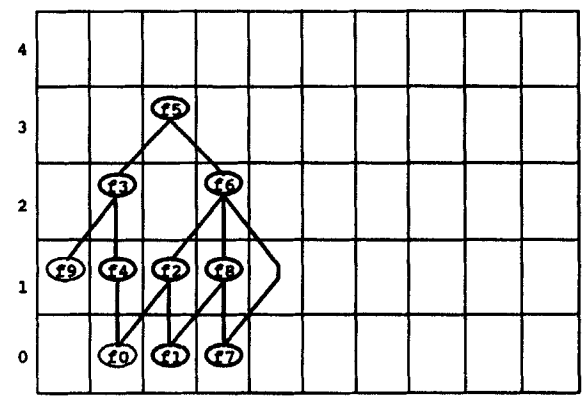

Figure 15. Finally, the nodes are sorted on each level to reduce the number of crossing edges between levels

concatenation of all layers). Layers correspond to functions and their data-flow (i.e., parameter passing, types, etc.). For each function, the algorithm partitions all related entities (i.e., variables, parameters, constants, types, etc.) and assigns them to their corresponding columns. All entities are then centred inside their layers with respect to the position of their related function. Notice that such a layered approach results in crossing-free layout views. The algorithm computes the position of each entity inside its corresponding column using the following formula:

$$\text{position} = y + ((max \text{ div } 2) \bmod 2) - n \text{ div } 2 + i \qquad (1)$$

where: y is the position of the highest entity in the layer, *max* is the maximum number of entities in any column of the layer, n refers to the number of entities in the same column and layer, and i is the ordinal number of the entity in that column.

For example, consider the second layer of the colonnade shown in Figure 16 (for simplicity we consider only five columns in this example). Suppose that we want to calculate the position of *parameter4*. The second layer begins at position $y1 = 4$ and has a $max = 3$ (there are no more than three entities in any column of the same layer). The parameter column contains three elements, so $n = 3$. Finally *parameter4* is the second entity in the layer, so $i = 2$. After replacing these values in the formula it becomes position = 4 + ((3 div 2) mod 2) − (3 div 2) + 2 = 6. Thus, *parameter4* is placed at the sixth position of its corresponding column in the colonnade layout.

3. A MAINTENANCE EXERCISE

In this section, we perform a maintenance exercise using the toolset. We are given a C++ program that manipulates strings of characters and stores them in a table. Our maintenance task is to modify the program in order to count and print the number of inserted strings. In addition, during the exercise we wish to locate and remove existing *dead* code (i.e., segments of code which are not used in the program).

First, we analyse the source code files comprising the C++ program using the toolset. The header files are parsed recursively following the *#include* statements and the database is populated with various dependencies found in the C++ code. Next, we display the file

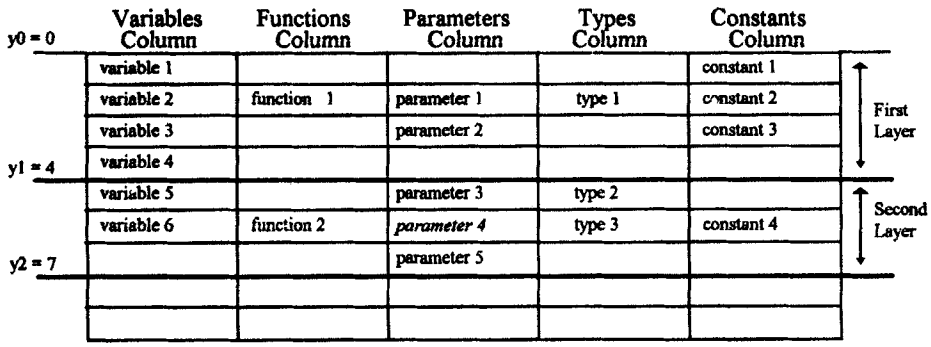

Figure 16. 'Parameter4' *is placed at the sixth position of the third column in the colonnade*

dependencies graph which is shown in Figure 17. From this graph, we can see that the C++ program consists of four source code files (i.e., .c) and three header files (i.e., .h). This information depicts the file structure of the program which helps us estimate the impact of code changes on the overall program architecture. For example, if we modify the header file *table.h*, as a consequence the source files *table.c* and *main.c* may also be affected.

In order to recover part of the design of the C++ program, we display the inheritance hierarchy graph as shown in Figure 18. From this figure we can see that this program contains three classes called *Name*, *Word* and *Table*. The class *Word* inherits properties

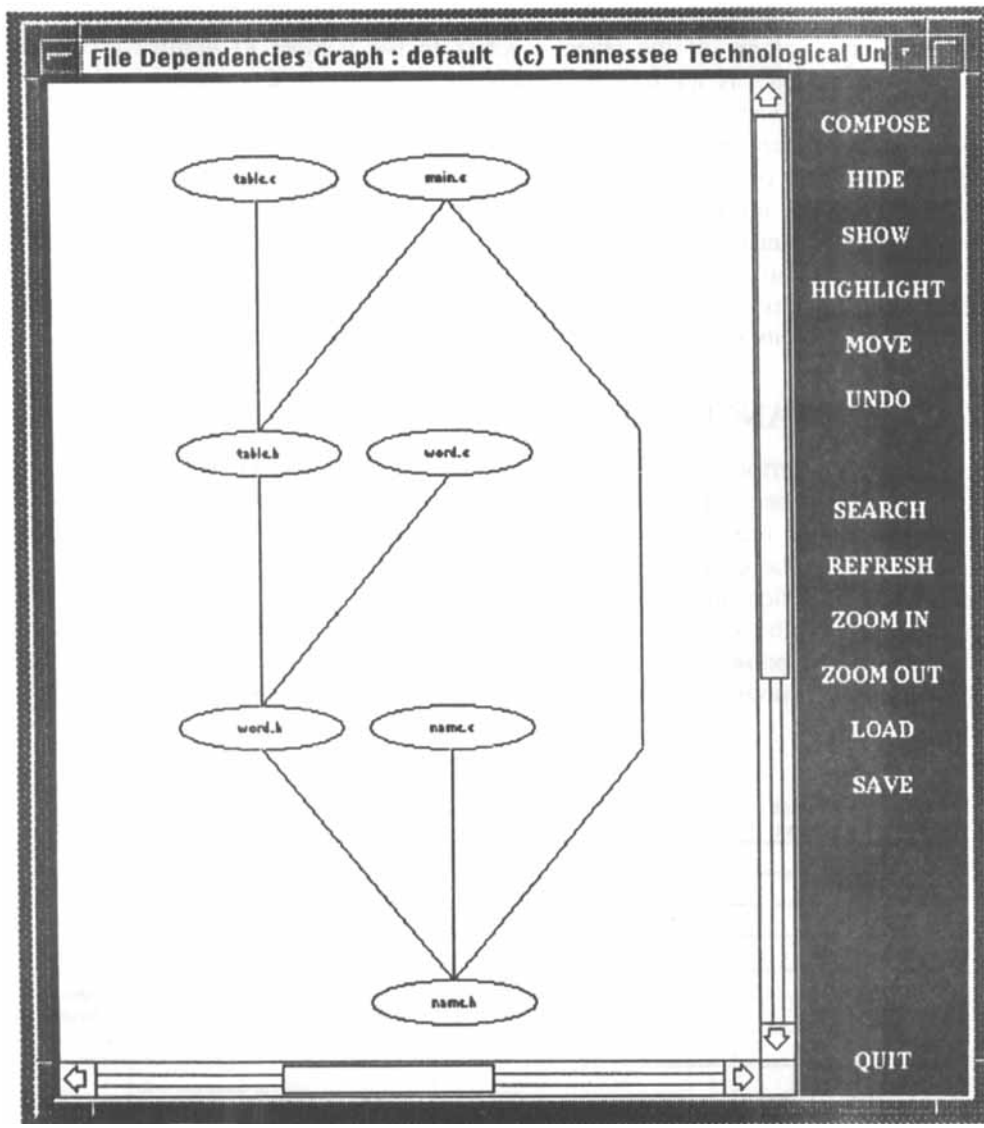

Figure 17. The file dependencies graph of a C++ program

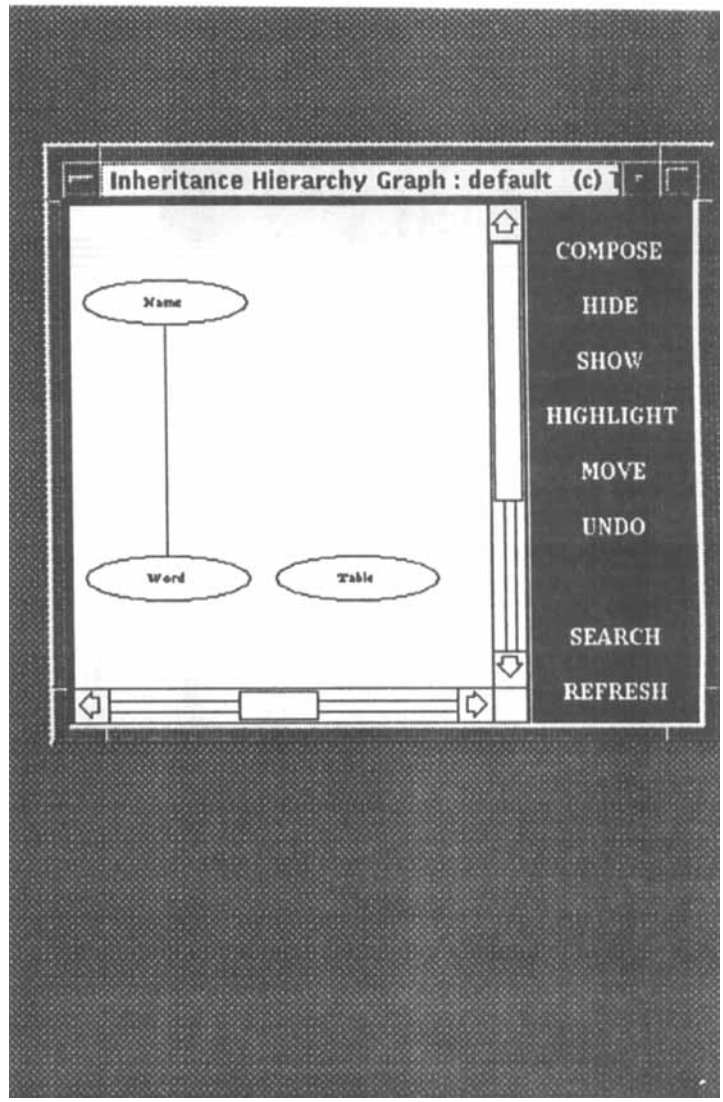

Figure 18. Inheritance hierarchy graph of a C++ program

from the class *Name*. Since we need to count the number of strings inserted in a *table*, it appears that the class called *Table* is a good place to start.

In order to retrieve more information about the class *Table* we visualize the data-flow program dependencies using the colonnade display. (In the case of a lengthy colonnade, a SEARCH facility is available which displays an alphabetical list of all functions in the program and allows the user to locate the desired one.) Searching through the colonnade, we locate one function called *insert* (see Figure 19). As we can see from the highlighted view in Figure 19, *insert* is a member function of the class *Table*. Moreover, it has one parameter called *N* which returns an address to a constant class called *Name*. In addition, the *insert* function uses a variable called *toInsert* which is defined as a pointer to a class

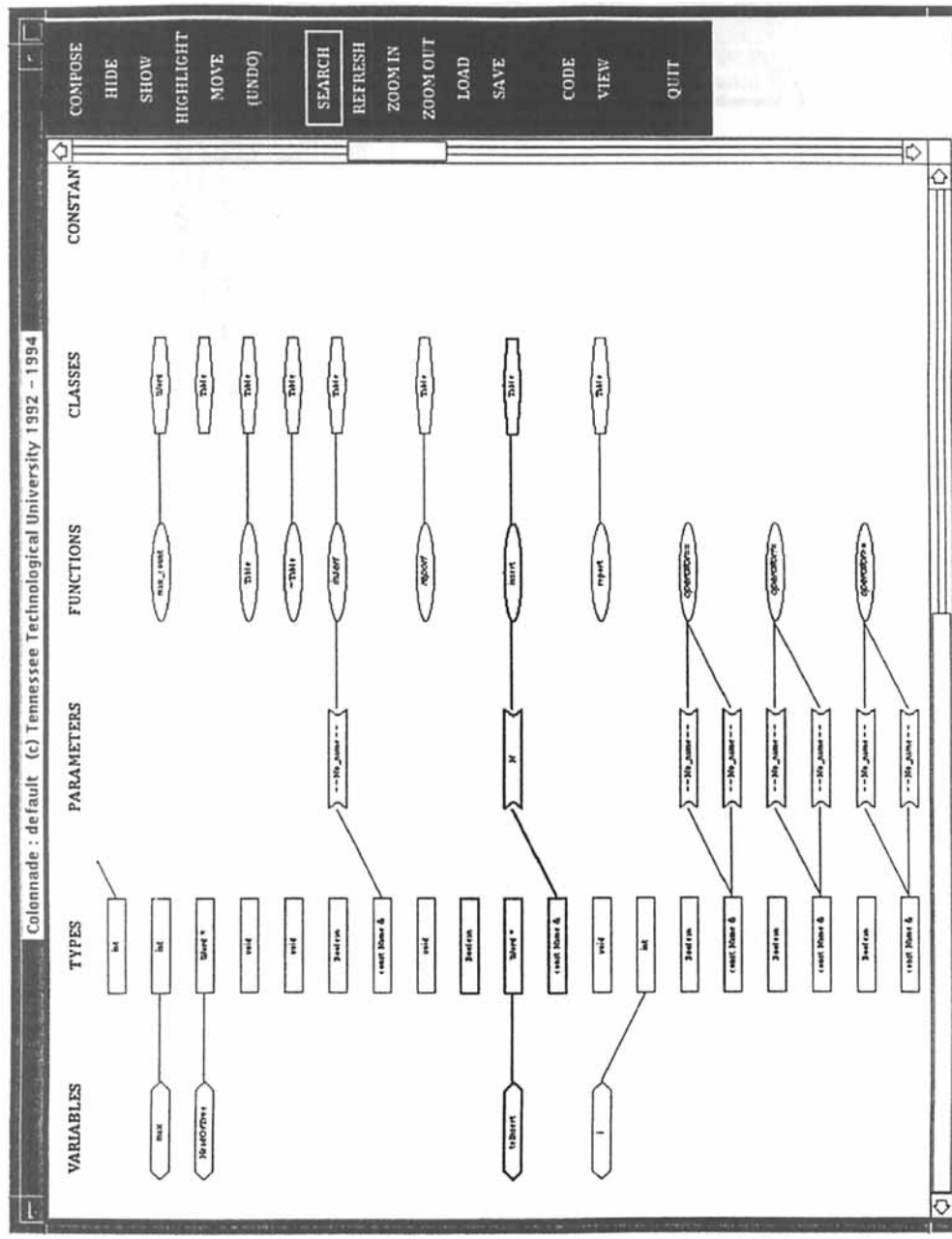

Figure 19. Data-flow information for the 'insert' function is shown in bold

called *Word*. Also, we can visualize the fact that the *insert* member function returns a boolean value (after moving the FUNCTIONS column next to the TYPES).

Next, we access the code of the *insert* function (by using the CODE option) and retrieve its body (i.e., definition) in a separate window (see Figure 20). Looking at its code we realize that this function is used to insert a *Name* object (i.e., a string of characters) into a data structure called *Tree* . Since this exercise asks us to keep track of the number of strings inserted in the table, we need to define a counter and increment it each time the *insert* function is called. Therefore, we add an extra line at the end of the *insert* function (using emacs or vi) in order to increment a new variable called *total* each time the function is called (see Figure 21).

Next, we need to declare the *total* variable. Using the VIEW command we display all the functions calling *insert* as well as the functions and member functions called by it (see Figure 22). As we can see from this figure, a user-defined function called *processfile* is the only one that calls *Table::insert*. In the same figure, we can observe that the *insert* function sends messages to two other member functions called *Word::operatornew* and *Word::put_in_tree*. This is an indication that the *total* variable must not be local to the *insert* member function because we want to access it at the end of the execution.

Therefore, we need to declare it inside the *processfile* function. To this end, we display its colonnade (see Figure 23). In this graphical view, we can see that *processfile* has one parameter called *fn* which returns a pointer to a constant character. In addition, it uses two data objects called *n* and *t* which are instances of the *Name* and *Table* classes respectively. Moreover, the *processfile* function returns no value (i.e., void). Finally, we access its source code where we declare the *total* variable. Moreover, we add an output statement in the body of *processfile* for displaying the number of strings inserted in a table (see highlighted line in Figure 24).

The first modification of the program is now completed but we still have to locate and remove any existing dead code (i.e., code segments that are not used). To this end, we display the control-flow graph which includes the call relations between functions. If a function is not fully connected in the control-flow graph (i.e., it does not call or is not called by any other function), this is an indication that the source code of such function may not be used by the program. Figure 25 shows the complete control-flow graph of our C++ program. It appears that this graph is disconnected in two places at the bottom of the layout. After zooming in, Figure 26 shows that a member function called *Name::operator+=* appears to be disconnected (highlighted box in Figure 26).

We access its code to see if it does not contain anything important for the rest of the program (see Figure 27). We soon realize by looking at the code that this is a definition of an operator which was provided with the *Name* class and which is not used in this program. Hence, for the purposes of this maintenance task, this code is considered dead and can be removed. Another function called *operator<<* appears to be disconnected as well (see highlighted oval in Figure 28). Again, by looking at its source code we discover that this function is used to display an object of class *Name* but it is not used in the program. Moreover, since the member function *Name::length* is only called by *operator<<* it can also be removed.

Finally, we have completed the required modification to the program and removed some unused code. Next, we reparse the modified file and update the database with the new information. Figure 29 shows the updated data-flow program dependencies about *pro-*

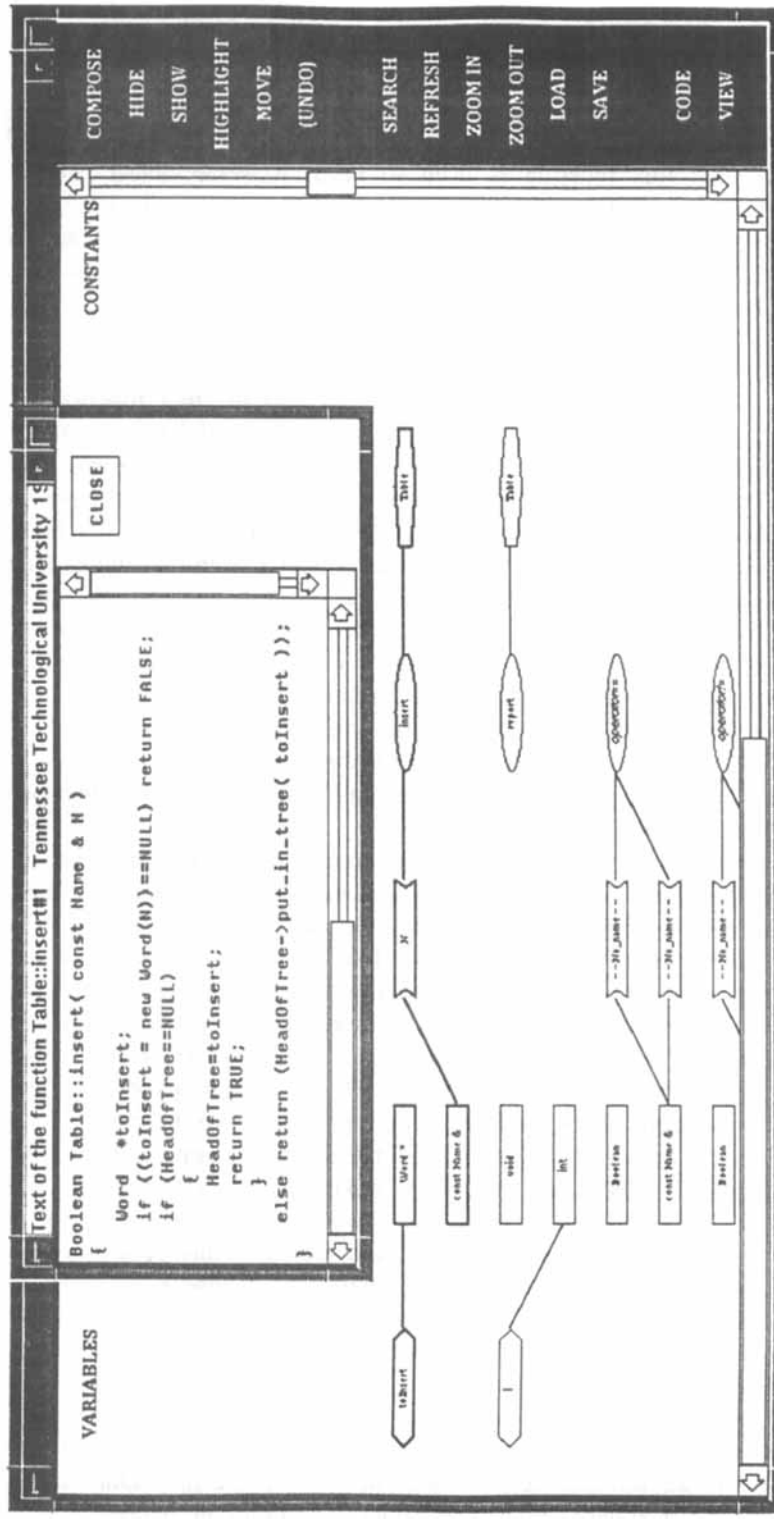

Figure 20. The body of the 'insert' *function is displayed*

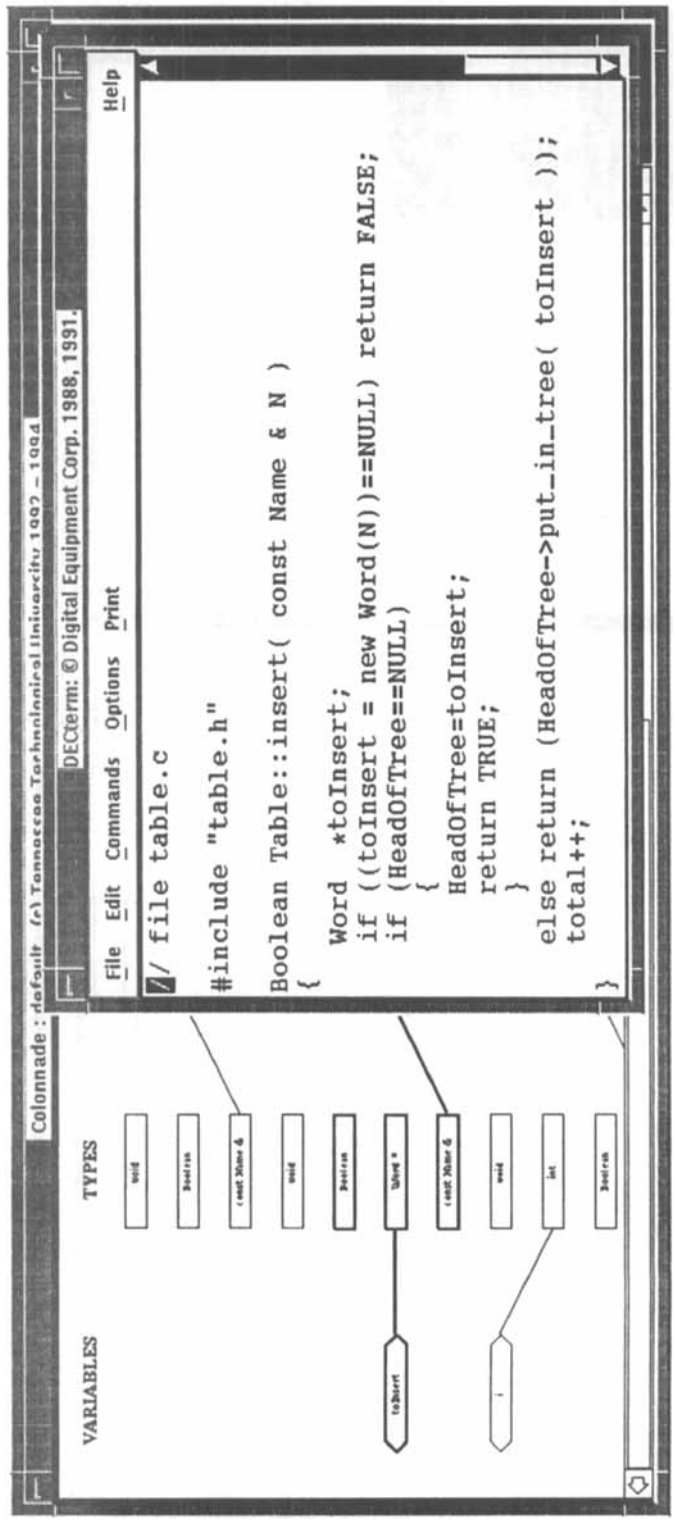

Figure 21. Modifying the source code of the 'insert member' *function*

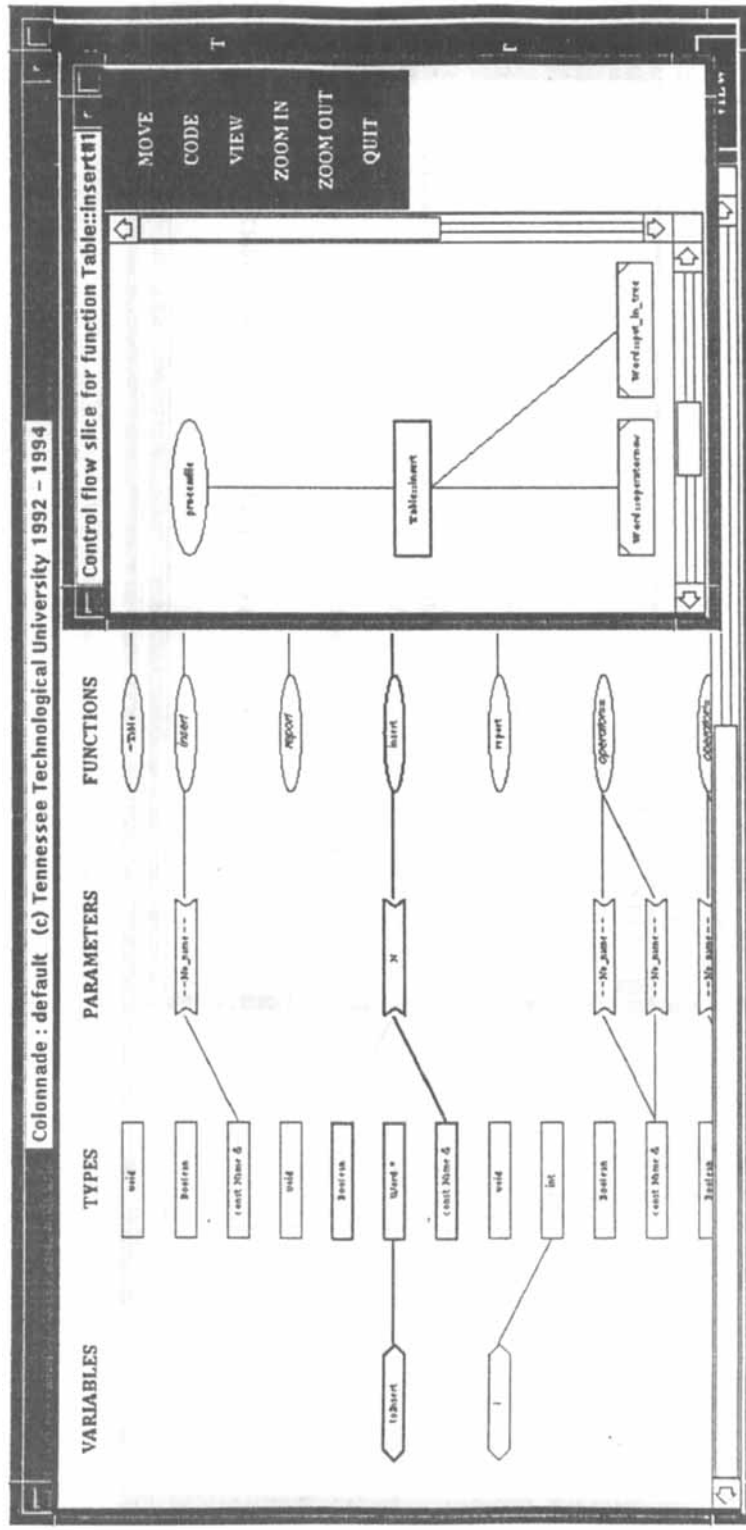

Figure 22. Control flow information for the 'insert' *function is displayed*

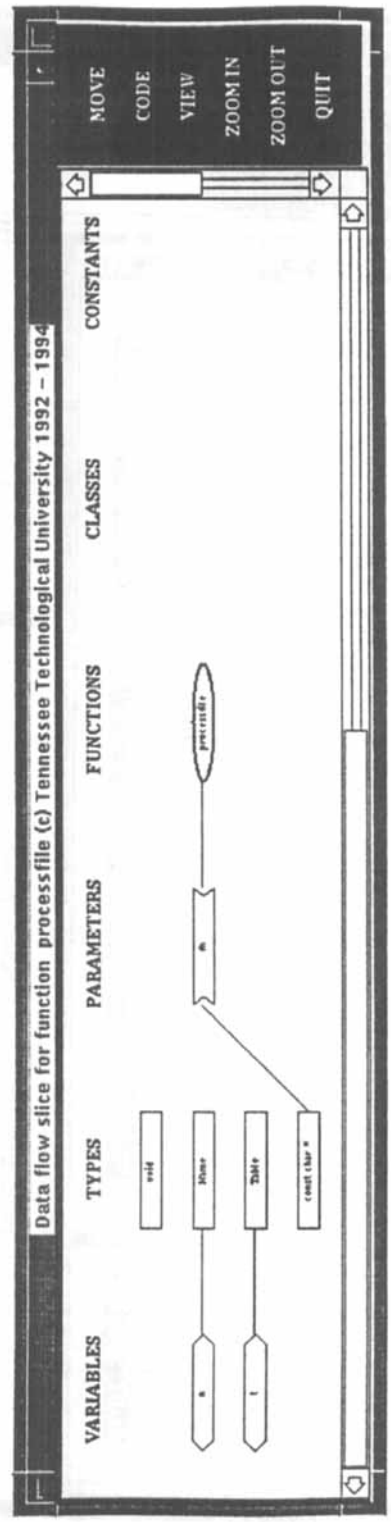

Figure 23. Data-flow information for the 'processfile' *function is displayed*

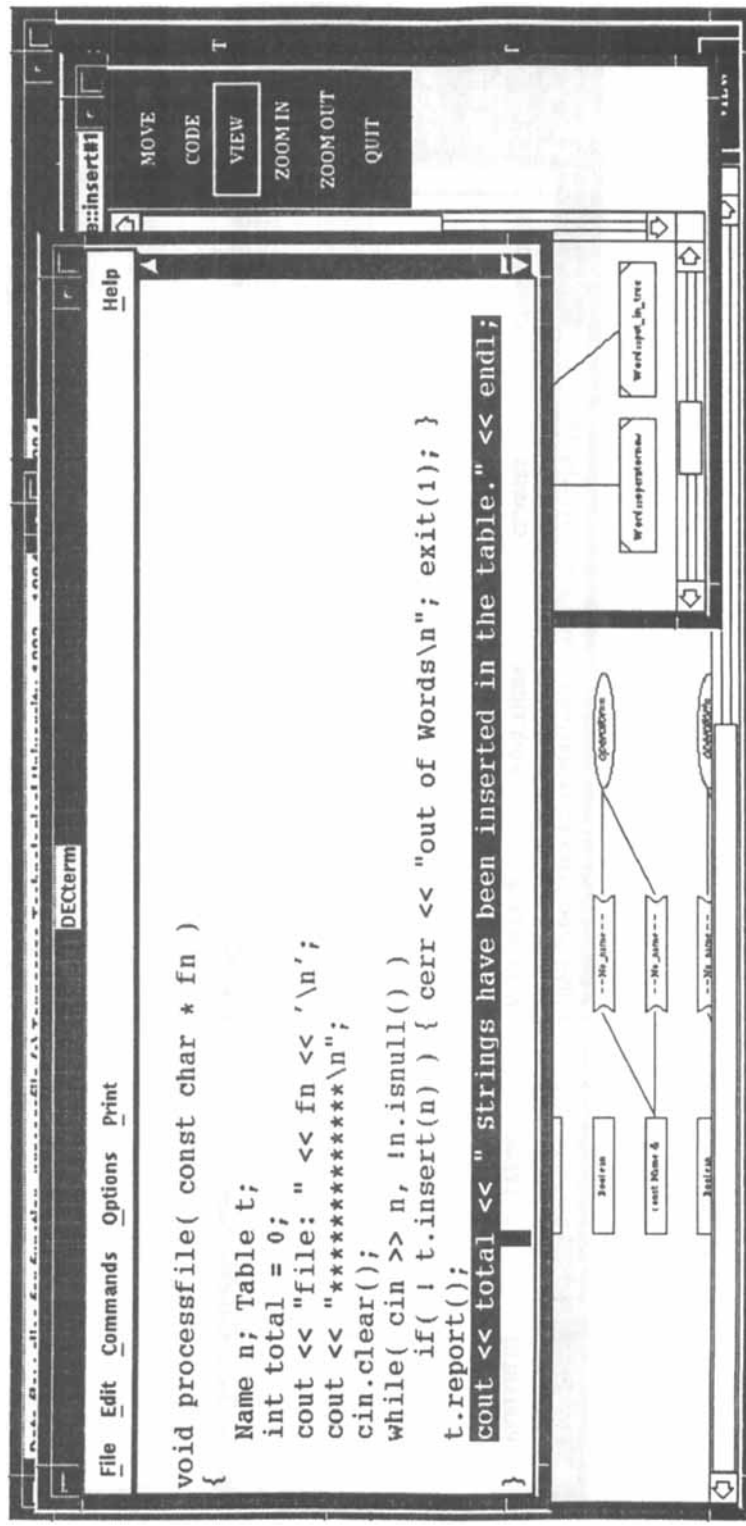

Figure 24. The source code of the 'processfile' *function is modified*

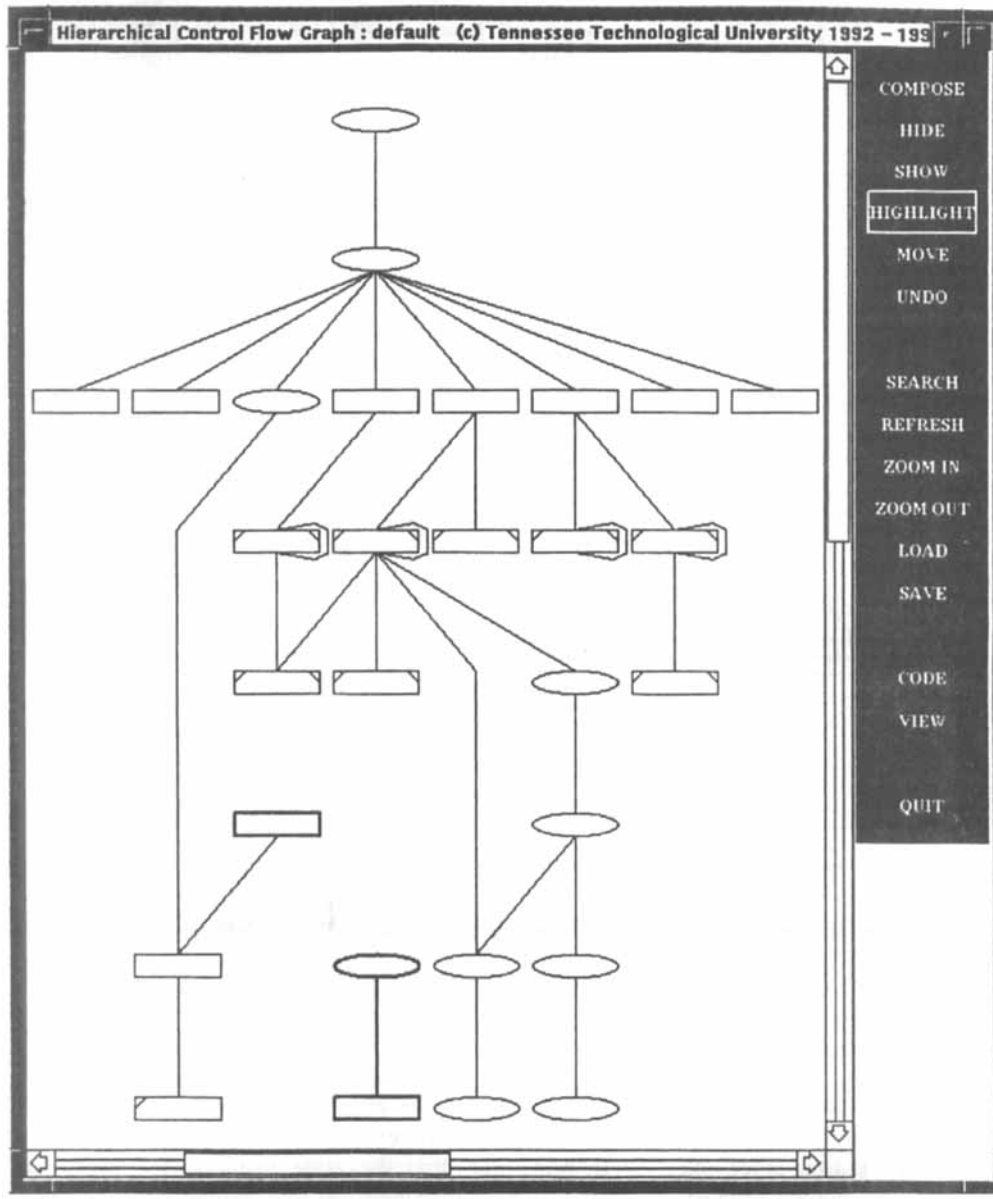

Figure 25. The control flow graph shows some unused functions

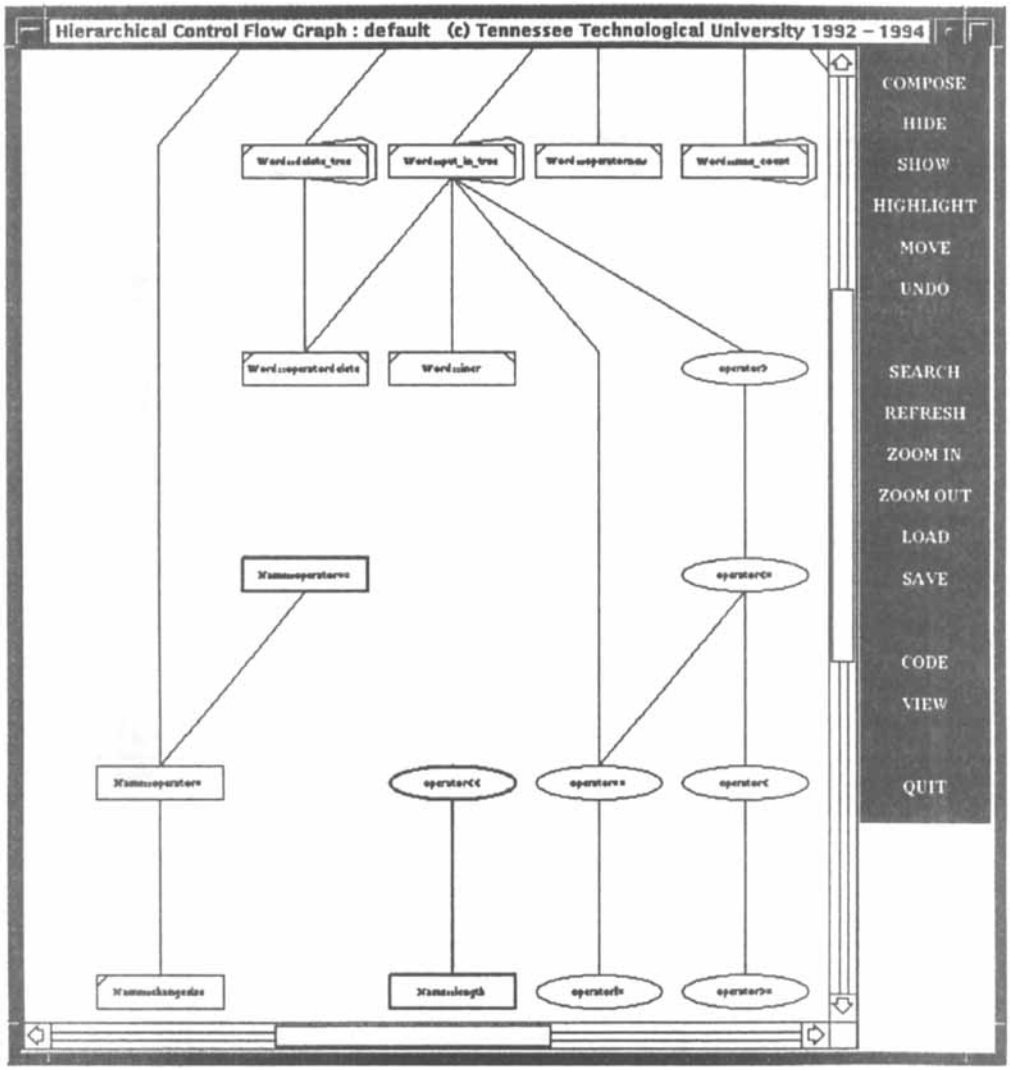

Figure 26. A candidate for dead code is shown in bold

cessfile. In this figure, the added variable called *total* and its data-type (i.e., int) is now displayed among the other local variables of the *processfile* function.

The above maintenance exercise demonstrates that the code visualization toolset used is a promising vehicle towards facilitating the comprehension and maintenance of existing hybrid C++ programs. Specifically, the visualization techniques, browsing features, and abstraction mechanisms provided by the toolset assist in understanding the combination of procedural and object–orientated programming techniques. Finally, the various graphical displays used facilitate program modifications by reducing the time spent in understanding the highly abstracted textual representation of hybrid C++ code.

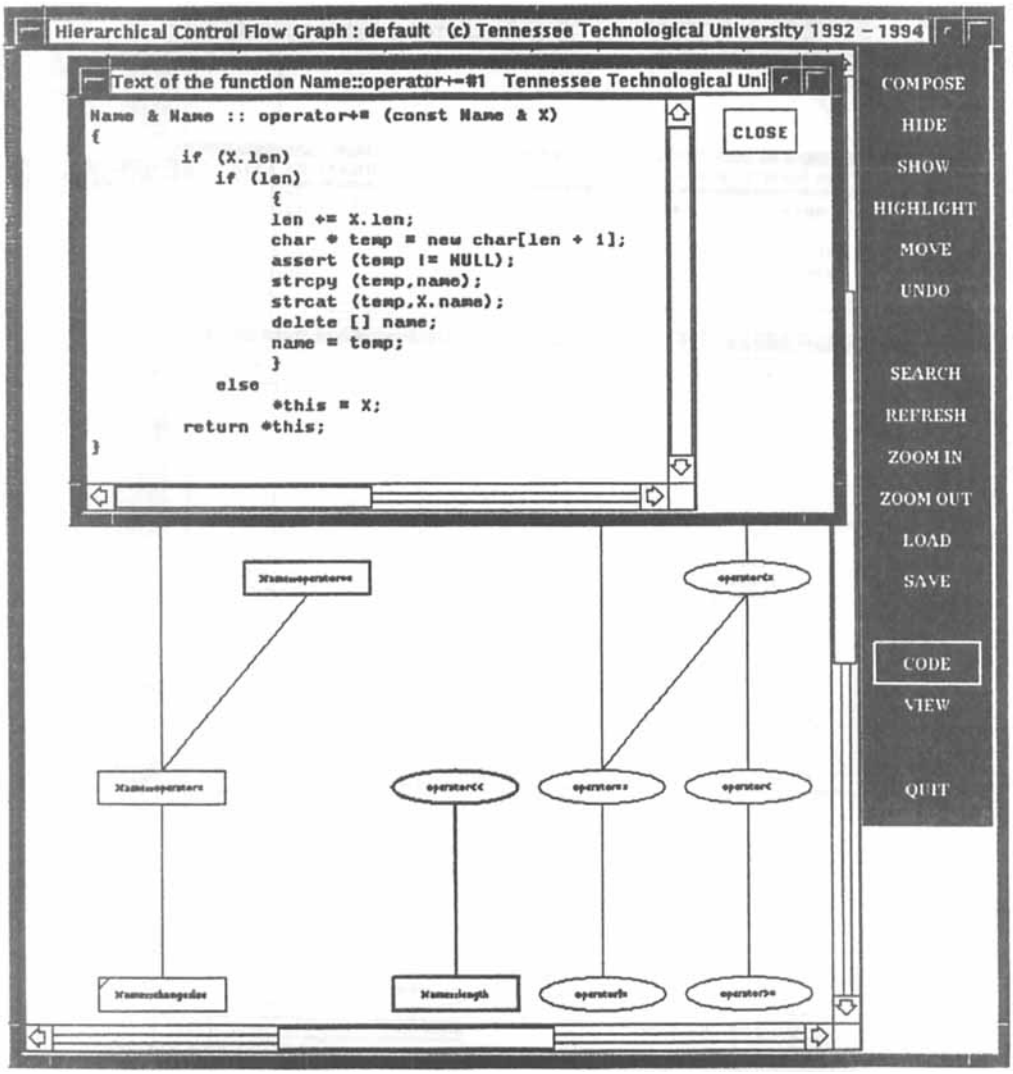

Figure 27. The source code of the 'Name::operator++' *member function is displayed*

4. EXPERIMENTS

Various experimental and empirical data have been gathered at the CARE Laboratory (http://www.csc.tntech.edu) during several program comprehension and maintenance exercises using the toolset. Specifically, we have shown experimentally that the use of the toolset has increased the users' productivity by 37·4% and improved the quality of the changes (i.e., less code statements are modified) when the tools are available (Linos *et al.*, 1994).

5. CONCLUSIONS

Currently, a prototype version of the toolset runs on DEC and HP workstations under the Unix operating system. It also supports an X-based Motif graphical user interface. The

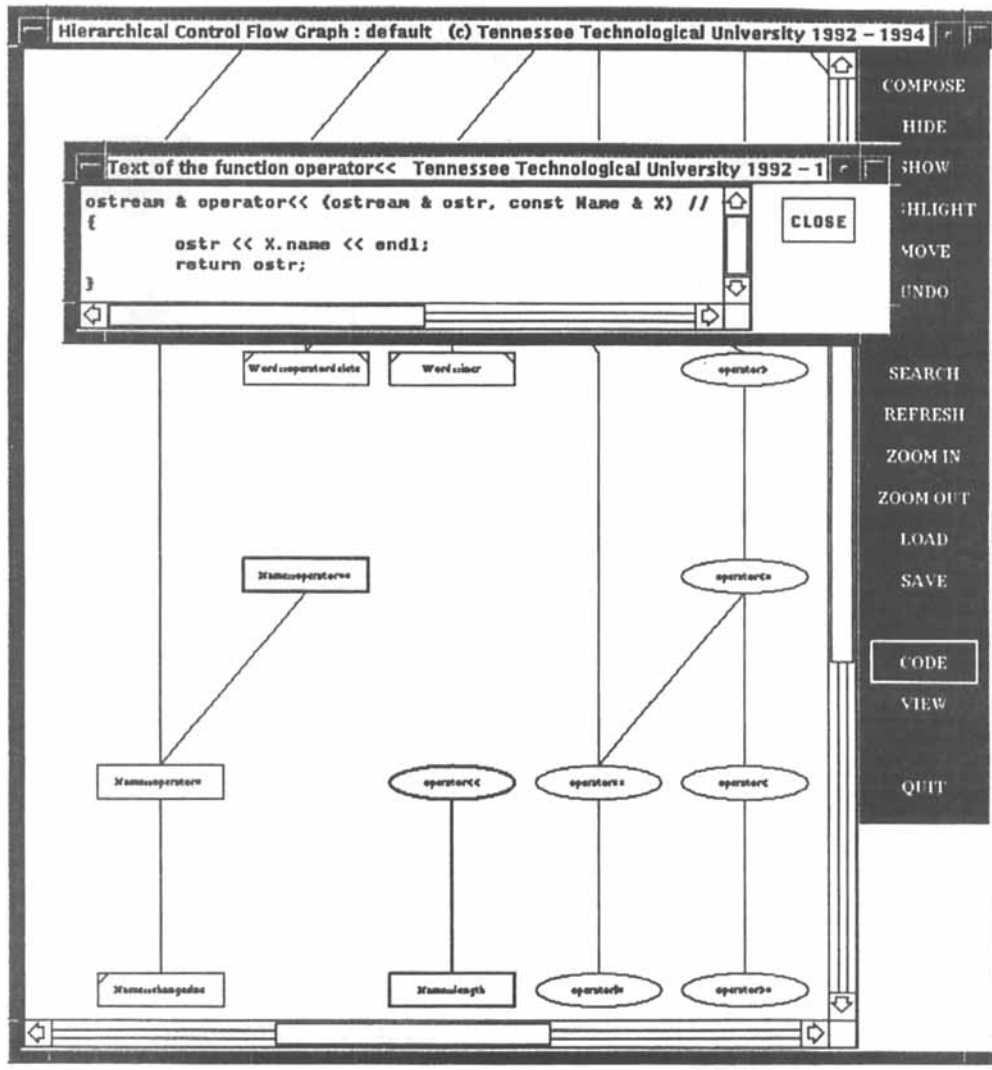

Figure 28. The source code of the 'operator<<' *function is displayed*

parsing facility of the dependency analyser is implemented using *lex* a lexical analyser available in the Unix environment (Levine, Mason and Brown, 1992, pp. 27–49, 147–179).

So far, several improvements have been suggested by the users of the prototype toolset. A *print screen* option within the toolset could be useful. Also, compiling and executing a modified program without exiting the toolset (by using *makefiles*) is another practical improvement. Moreover, some additional scale factors could allow the display of very large graphs and make the textual labelling more readable. The addition of colours could improve the readability of the call-graphs and the colonnade. For example, vivid colours could be used with the graphical displays in order to prominently show some important object-orientated program dependencies such as *dynamic, implicit* or *polymorphic* dependencies.

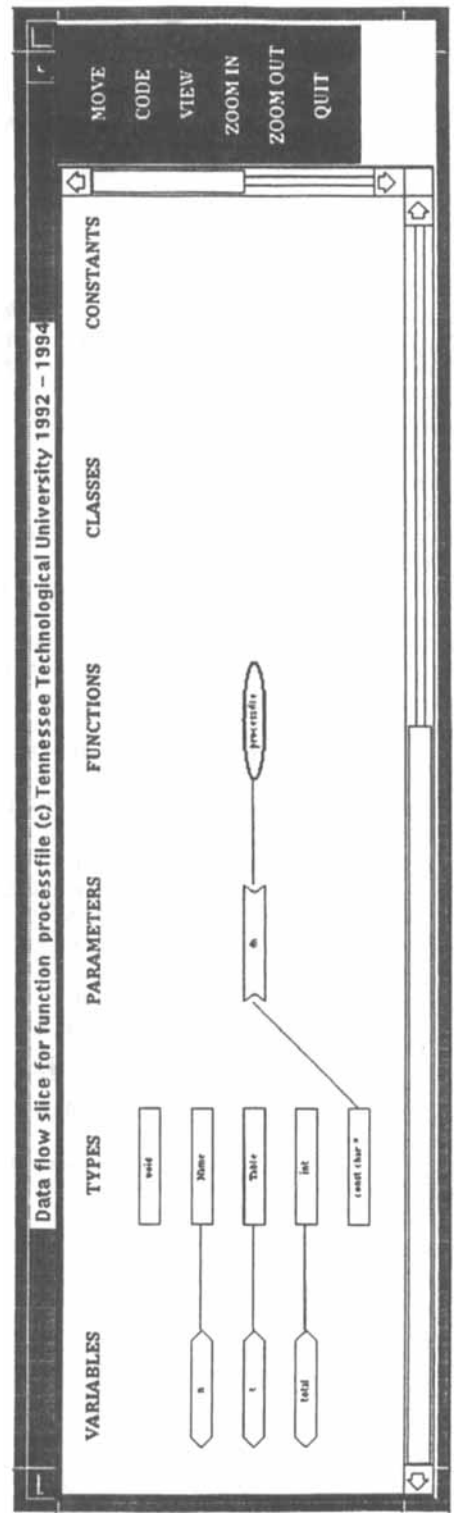

Figure 29. The updated data–flow graph for the 'processfile' *function*

Acknowledgements

We are grateful to Professor Leland Long, the Chairperson of the Computer Science Department at Tennessee Technological University, for his continuous encouragement and support. We also thank the reviewers and Editor of the *Journal* for their valuable comments.

References

Chapin, N. (1988) 'Software maintenance life cycle', *Proceedings of the Conference of Software Maintenance—1998*, IEEE Computer Society, Los Alamitos, CA, pp. 6–13.

Chen, Y.-F. and Grass, J. E. (1990) 'The C++ information abstractor', *Proceedings of the Second C++ Conference*, USENIX, San Francisco, CA, AT&T Bell Labs, Berkeley, CA, pp. 34–50.

Choi, S. C. and Scacchi, W. (1990) 'Extracting and restructuring the design of large systems', *IEEE Software*, **17**(1), 66–71.

De Pauw, W., Helm, R., Kimelman, D. and Vlissides, J. (1993) 'Visualizing the behavior of object-oriented systems', *SIGPLAN Notices*, **28**(10), 326–337.

Edelstein, D. V. (1993) 'Report on the IEEE STD–1219—1993 standard for software maintenance', *Software Engineering Notes*, **18**(4), 94–95.

Lejter, M., Meyer, S. and Reiss, S. P. (1992) 'Support for maintaining object-oriented programs', *Transactions on Software Engineering*, **18**(12), 1045–1052.

Levine, J., Mason, T. and Brown, D. (1992) *Lex & Yacc*, O'Reilly & Associates, Inc., New York, NY.

Linos, P. K., Rajlich, V. and Korel, B. (1991) 'Layout heuristics for graphical representations of programs', *Proceedings of the Conference on Man, Machine and Cybernetics*, Charlottesville, Virginia, USA, IEEE Computer Society Press, Los Alamitos, CA, pp. 1127–1132.

Linos, P. K., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P. and Tulula, P. (1993) 'CARE: an environment for understanding and re-engineering C programs', *Proceedings of the International Conference on Software Maintenance—1993*, Montreal, Canada, IEEE Computer Society Press, Los Alamitos, CA, pp. 130–139.

Linos, P. K., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P. and Tulula, P. (1994) 'Visualizing program dependencies: an experimental study', *Software: Practice and Experience*, **24**(4), 387–403.

Linos, P. K. and Courtois, V. (1994) 'A tool for understanding object-oriented program dependencies', *3rd Workshop on Program Comprehension*, Washington, DC., USA, IEEE Computer Society Press, Los Alamitos, CA, pp. 20–27.

Sametinger, J. (1990) 'A tool for the maintenance of C++ programs', *Proceedings of the Conference on Software Maintenance—1990*, San Diego, CA, USA, IEEE Computer Society Press, Los Alamitos, CA, pp. 54–59.

Sharon, D. (1996) 'Meeting the challenge of software maintenance', *IEEE Software*, **13**(1), 122–125.

Taenzer, D., Ganti, M. and Podar, S. (1989) 'Object-oriented software reuse: the yoyo problem', *Journal of Object-Oriented Programming*, **2**(5), 30–35.

Wilde, N. and Huitt, R. (1992) 'Maintenance support for object-oriented programs', *Transactions on Software Engineering*, **18**(12), 1038–1044.

Authors' biographies:

Panagiotis K. Linos focuses his research on developing tools for the comprehension and maintenance of complex software. During 1994–1996, he was involved in the Software Cost Reduction project at the Naval Research Laboratory in Washington, DC. Dr. Linos is an Associate Professor of Computer Science at Tennessee Technological University. His B.Sc. is in Mathematics from the University of Athens, Greece, and his M.Sc. and Ph.D. are in Computer Science from Wayne State University in Michigan, U.S.A.

Vincent Courtois focuses on parsing techniques for program analysis and graphical user interfaces for software maintenance tools. He worked on the CARE project at Tennessee Technological University during 1993–1994. Currently, he is working at RESOCOM Services, 17 rue St. Florentin, 75008 Paris France. Mr. Courtois holds an engineering diploma from Hautes Etudes Industrielles in Lille, France.